GPU-Accelerated Maximal Quasi-Clique Mining

Michael Greenbaum
Rowan University
greenb88@students.rowan.edu

Ke Fan
Temple University
ke.fan@temple.edu

Wajid Manzoor
Rowan University
wajidm36@students.rowan.edu

Guimu Guo
Rowan University
guog@rowan.edu

***Abstract*—The exploration of maximal quasi-cliques (MQC) within graphs is a computationally intensive problem (NP-hard) with wide-ranging applications in social network analysis, bioinformatics, and network security. The massive parallelism in graphics processing units (GPUs) is well suited for solving maximal quasi-cliques mining. However, it is a challenge to parallelize the quasi-clique algorithm on the GPU due to (i) the redesign of 6 pruning rules by utilizing a GPU-aware data structure following coalesced memory access, (ii) the enormous memory requirement of current approaches, and (iii) the drastic load imbalance among different tasks and the efficient utilization of threads and blocks. In this paper, we propose cuQC, the first GPU-accelerated approach designed to efficiently mine MQC, leveraging the parallel processing capabilities of modern GPUs. The cornerstone of our approach is developing an innovative data structure optimized for GPU memory hierarchy, facilitating rapid access and manipulation of graph data. Our experimental evaluation demonstrates the efficacy of our approach by comparing the execution time and graph size that can be handled against the state-of-the-art CPU implementations. Our source code is released at https://github.com/Mike12041204/cuQC.**

***Index Terms*—Graph Mining, Parallel Computing, CUDA**

I. INTRODUCTION

The discovery of dense subgraphs from one big graph has attracted increasing attention. One notable dense structure is γ-quasi-clique, which is a natural generalization of a clique that is useful in mining various networks [1]. Specifically, given a degree threshold γ and an graph G, a γ-quasi-clique is a subgraph of G, denoted by $g = (V_g, E_g)$, where each vertex connects to at least $\lceil \gamma \cdot (|V_g| - 1) \rceil$ other vertices in g [2], [3].

Maximal quasi-cliques (MQC) are critical in graph analysis. A MQC is a specific type of γ-quasi-clique that cannot be extended further by adding more vertices from the graph while maintaining its density threshold γ. Enumerating all MQCs within a graph that meet or exceed a specified minimum vertex count threshold τ_{size} is a significant computational challenge.

MQC mining is extensively applied in a variety of domains, such as identifying functional modules or protein complexes in biological networks [4], [5], uncovering communities within social networks [6], [7], identifying spam/phishing email sources [8], [9].

Challenges and Existing Methods. The problem of finding all MQCs in a graph is NP-Hard [10], [11]. Even determining whether a given quasi-clique is maximal is already NP-hard. This high computational complexity arises from the need to explore potentially vast numbers of vertex subsets to identify those that satisfy the quasi-clique criteria. This becomes increasingly challenging as graphs can be extremely large in real-world applications.

Several algorithms have been proposed for mining MQC, including Crochet [12], [13], Cocain [14], and Quick [3]. These algorithms generally use a depth-first order to explore the search space (i.e., the set of all possible vertex sets), incorporating pruning techniques that eliminate vertices if they cannot possibly form a quasi-clique satisfying the density criteria. Despite sophisticated pruning techniques, these state-of-the-art algorithms [3], [12], [14] are unable to scale to large datasets. For example, Quick [3] can only handle graphs with thousands of vertices. This limitation leads quasi-clique mining research to focus on developing heuristic algorithms that can provide practical solutions under specific conditions [10].

Since MQC mining involves exploring numerous combinations of vertices and edges, following the divide and conquer paradigm, the problem of mining a big graph can be partitioned into tasks that mine smaller subgraphs concurrently. Owing to the massive parallel processing capabilities, high bandwidth, and low power requirements, GPUs are well suited for solving the MQC mining problem in big graphs, where the sheer volume of data and the number of potential subgraphs to analyze can be overwhelming for traditional CPUs. However, achieving high performance for mining MQCs on GPUs presents several challenges:

- *Irregular Memory Access Patterns*: Operations like diameter-based pruning (see Section II), and vertex degree updates, require intersecting vertex and neighbor sets. Since neighbors may not reside in contiguous memory, these intersections involve irregular, non-sequential memory access—making them the program's primary performance bottleneck.
- *Memory Limitations*: GPUs have limited onboard memory, which can pose a significant constraint when dealing with big graphs. Besides, effectively managing the GPU's different memory types—global, shared, and registers—is challenging yet crucial for performance optimization, given their varying sizes, speeds, and access patterns.
- *Highly Imbalanced Workload*: The node number in each subgraph and each node's degree can vary significantly, causing a workload imbalance. Addressing these imbalances requires multi-level solutions, spanning warps, thread blocks, and grids, further complicating the issue.

Our **contributions** are summarized below:

- We propose a high-throughput MQC algorithm designed

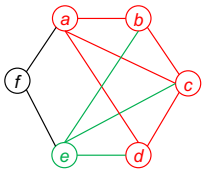


Fig. 1. An Illustrative Graph

for parallel execution, incorporating various pruning strategies optimized for the GPU architecture.

- We introduce novel task data structures to facilitate highly concurrent memory accesses. A dynamic task scheduling approach and a warp-level intersection technique ensure balanced workload distribution. A task cache mechanism has been devised to control the tasks spawning rate, ensuring memory utilization remains bounded.
- We employ optimization techniques to amplify effectiveness on GPU, including a hybrid CPU-GPU approach, a shared memory buffer, and a single-pass expansion procedure.
- We developed a distributed memory version of the algorithm that leverages multiple GPUs, enabling the handling of larger graphs at enhanced processing speeds.
- We compare the above algorithms against the state-of-the-art approaches comprehensively, using 21 public graph datasets of various characteristics.

II. PRELIMINARIES

A. Definitions

Graph Notations. We consider an undirected graph $G = (V, E)$, where V (resp. E) is the set of vertices (resp. edges). We denote the vertex set as $V(G)$. Given $S \subseteq V$, we use $G(S)$ to denote the subgraph of a graph G induced by S, which includes a subset of the vertices of $V(S)$ together with any edges whose endpoints are all in this subset. $|S|$ is the number of vertices in S.

Given $v \in G$, $N(v)$ denotes the set of neighbors of v in V. We further define $d(v) = |N(v)|$ as the degree of v in G. Given a vertex subset $V' \subseteq V$, we define $N_{V'}(v) = \{u \,|\, (u, v) \in E, u \in V'\}$, i.e., $N_{V'}(v)$ is the set of v's neighbors inside V', and we also define $d_{V'}(v) = |N_{V'}(v)|$.

Problem Definition. Next, we formally define our problem.

Definition 1 (γ-quasi-clique): A graph $G = (V, E)$ is a γ-quasi-clique ($0 < \gamma \leq 1$) if G is connected, and for every vertex $v \in V$, its degree $d(v) \geq \lceil \gamma \cdot (|V| - 1) \rceil$.

Definition 2 (Maximal γ-quasi-clique): A γ-quasi-clique $G(S)$ is maximal if and only if there is no other γ-quasi-clique $G(S')$ containing $G(S)$, i.e., $S \subset S'$.

To illustrate using Fig. 1, consider $S_1 = \{v_a, v_b, v_c, v_d\}$ (i.e., vertices in red) and $S_2 = S_1 \cup v_e$. If we set $\gamma = 0.6$, then both S_1 and S_2 are γ-quasi-cliques: every vertex in S_1 has at least 2 neighbors in $G(S_1)$ among the other 3 vertices (and $2/3 > 0.6$), while every vertex in S_2 has at least 3 neighbors in $G(S_2)$ among the other 4 vertices (and $3/4 > 0.6$). Also, since $S_1 \subset S_2$, $G(S_1)$ is not a maximal γ-quasi-clique. But graph G, as a whole, is not a 0.6-quasi-clique as v_f only has 2 neighbors among the other 5 vertices ($2/5 < 0.6$). If a graph is a γ-quasi-clique, its subgraphs usually become uninteresting, so we only mine maximal γ-quasi-cliques in this paper.

Small γ-quasi-cliques are also trivial and not interesting. For example, a single vertex or an edge with two end-vertices are quasi-cliques for any γ. In the literature of dense subgraph mining, researchers usually only strive to find large subgraphs. These state-of-the-art algorithms [2], [3] used the minimum size threshold τ_{size} to filter small quasi-cliques.

Definition 3 (Problem Statement): Given a graph $G = (V, E)$, a minimum degree threshold $\gamma \in (0, 1]$ and a minimum size threshold τ_{size}, we aim to find all the vertex sets S such that $G(S)$ is a maximal γ-quasi-clique of G, and that $|S| \geq \tau_{size}$.

For a smaller value of γ, there exist numerous γ-quasi-cliques, yet the majority of them are of small size and not cohesive. γ-quasi-clique follow the property that for $\gamma \geq 0.5$, the diameter of a γ-quasi-clique is at most 2 [12]. In previous studies [2], [3], [13], it was a convention to set the $\gamma \geq 0.5$, although it is also possible to calculate the diameter's limit when $\gamma < 0.5$ using the Theorem 1 in [12]. This property can be used to remove vertices early from the candidate set that are not 2-hops away from the vertices already in the result set. Following [2], [3], [13], we focus on those γ-quasi-clique with $\gamma \geq 0.5$ only in this paper.

B. GPU Architecture

The nature of MQC mining involves exploring large numbers of potential subgraphs, which can be handled in parallel. GPUs are ideal for iterative and data-intensive tasks involved in MQC mining.

Streaming Processors. In GPU architecture, 32 threads form a warp, executing instructions uniformly. In particular, the GPU executes one warp of threads in a Single Instruction Multiple Data (SIMD) fashion. Warps compose a thread block assigned to a streaming multiprocessor (SM) equipped with execution units, L1 cache/shared memory, and registers. A GPU contains several SMs, and each SM contains hundreds of CUDA cores.

GPU Memory Architecture. Registers are the fastest in the GPU memory hierarchy and are allocated per thread. Threads in a warp can efficiently swap data using warp-level primitives that utilize registers. A shared L1 cache/memory is accessible to threads within the same block and is programmable but with limited space. It is best to keep elements shared by the thread block in shared memory and thread-local data in registers. The entire GPU uses a shared L2 cache, with global memory atop it, which has the slowest access rate but is accessible by all threads. GPU threads cannot directly access CPU memory, emphasizing the importance of CPU-GPU data movement and memory management. Maximizing data reuse on the GPU for tasks like MQC mining is a significant challenge, but efficiency improves if all necessary data for threads in a warp is obtained in a single memory transaction. This pattern, called *coalesced memory access*, occurs when all threads in the warp access consecutive memory addresses

CUDA programming. CUDA (Compute Unified Device Architecture) is a parallel computing platform and pro-

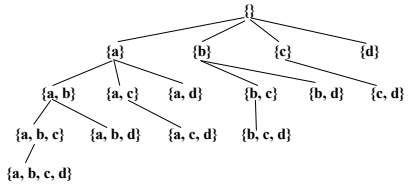


Fig. 2. Set-Enumeration Tree

gramming model developed by NVIDIA for general computing on its GPUs. A kernel function launched by the GPU is specified in the form *kernel_function*$<<<BLK\_NUM, BLK\_DIM>>>$, where *BLK_NUM* is the number of thread blocks and *BLK_DIM* is the number of threads in each thread block. The GPU allows multiple threads to execute the same code specified by the body of the kernel function, but on different data. This data parallelism is realized because each thread has access to the following built-in variables in CUDA:

- *blockIdx.x:* the index of a block;
- *blockDim.x:* the number of threads in each block, aka. the block dimension, as specified by *BLK_DIM*;
- *threadIdx.x:* the thread index within a block;

In this paper, we generalize those built-in variables as below.

- For a particular thread, its warp ID in the block: *WARP_ID* = *threadIdx.x* / 32
- The ID of a thread in the warp: *LANE_ID* = *threadIdx.x* % 32

III. RELATED WORK

A. The State-of-the-art Serial Algorithm

Existing research consistently utilizes a branch-and-bound (BB) strategy to enumerate MQCs. Their primary objective is to develop efficient pruning rules that narrow search space. The earliest BB algorithms proposed for MQC mining are Crochet [12], [13] and Cocain [14]. Subsequently, Quick [3] incorporated all prior pruning rules with additional new pruning techniques, like upper-bound and lower-bound pruning.

We will take Quick as an example to provide a brief introduction to the BB algorithm. The giant search space of a graph $G = (V, E)$, i.e., V's power set, can be organized as a set-enumeration tree [3]. Fig. 2 shows the set-enumeration tree T for a graph G with four vertices $\{a, b, c, d\}$ where $a < b < c < d$ (ordered by ID). Each tree node represents a vertex set S, and only vertices larger than the largest vertex in S are used to extend S. For example, in Fig. 2, node $\{a, c\}$ can be extended with d but not b as $b < c$; in fact, $\{a, b, c\}$ is obtained by extending $\{a, b\}$ with c. It has $ext(S) \subseteq (V - S)$ keep those vertices that can extend S further into a γ-quasi-clique. Many vertices cannot form a γ-quasi-clique together with S and can thus be safely pruned from $ext(S)$; therefore, $ext(S)$ is usually much smaller than $(V - S)$.

During the recursive branching process, Quick applies two types of pruning techniques, namely Type I pruning rules and Type II pruning rules. Type I pruning rules are conducted on $ext(S)$ and aim to refine $ext(S)$ by removing those vertices that satisfy certain conditions; Type II pruning rules are conducted on S and aim to prune those branches where vertices in S satisfy certain conditions. The rationale is that if a vertex v satisfies certain conditions, each MQC covered by this branch should not include this vertex. Thus, we can either remove v from $ext(S)$ for this branch, i.e., Type I pruning rules apply (if $v \in ext(S)$), or prune the entire branch, i.e., Type II pruning rules apply (if $v \in S$). For simplicity, we omit the details of these pruning techniques and refer to [2].

B. Parallel Solution

Quick [3] was only tested on small graphs: one with 4,932 vertices and 17,201 edges, and the other with 1,846 vertices and 5,929 edges. To scale to big graphs, parallel computing techniques have been widely used to solve the graph mining problem. Several subgraph-centric systems have been proposed for graph mining problems, including NScale [15], Arabesque [16], G-Miner [17] and T-thinker [18].

For example, building upon the parallel graph mining system T-thinker [18], Quick+ [2] has been developed to address the problem of mining maximal cliques. It recursively partitions the search tree in Fig. 2 into multiple subtrees via branching. Let us denote T_S as the subtree of the set-enumeration tree T rooted at a node with set S. Then, T_S represents a search space for all possible γ-quasi-cliques that contain all vertices in S. In other words, let Q be a γ-quasi-clique found by T_S, then $Q \supseteq S$. Note that the mining of T_S can be recursively decomposed into the mining of subtrees rooted in the children of node S in T_S, denoted by $S' \supset S$. Since $ext(S') \subset ext(S)$, the subgraph induced by nodes of a child task $\langle S', ext(S')\rangle$ is smaller.

Despite these advancements in parallel CPU systems, GPU-based solutions for MQC mining remain unexplored. This paper introduces the first GPU-accelerated approach to confront this challenge.

IV. DESIGN OF CUQC

A. Overview and Challenges

Task Based Design. Our algorithm follows the branch-and-bound algorithm framework of Quick+ [2] as introduced in Section III-A. We partition the set-enumeration tree and each subtree is encapsulated as a task, presented as a pair $\langle S, ext(S)\rangle$. Thus, every task can be accessed independently, enabling parallelism that was not possible in Quick's [3] recursive approach. Note that in Fig. 3, for simplicity, only the vertices in S are used to represent the tasks. In each level of the enumeration tree, tasks are stored consecutively in a *TaskList* and differentiated using various highlights in Fig. 3.

The first challenge is to determine the proper smallest computing unit for parallelism. In the parallel CPU program, Quick+ on T-thinker [18], the computing unit is a CPU thread. On the GPU, a warp consisting of 32 threads is a more suitable choice as it facilitates coalesced memory access of adjacency lists. The majority of operations in an MQC mining task involve the examination of vertex sets or the intersection of adjacency lists (to be described in Section IV-C). As shown in Fig. 3, we assign each warp to handle one task (which

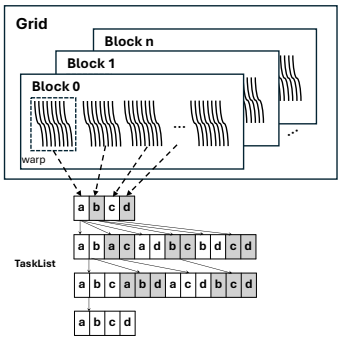


Fig. 3. Warp Task Assignment

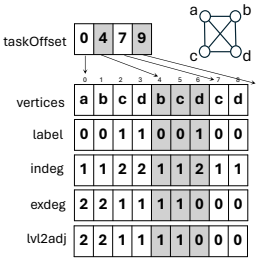


Fig. 4. Task Structure

corresponds to a node in the set-enumeration tree shown in Fig. 2). Every thread in the warp runs the related operations in parallel, such as updating the degree for all v's neighbors when moving v from candidate set $ext(S)$ to S. This representation and generation of numerous independent warp-oriented tasks allows for effective utilization of the GPU.

B. Data Structure

Graph Data Structure. We store the graph $G = (V, E)$ in the global memory compactly using a compressed sparse row (CSR) format [19], [20]. Please refer to Figure 2 in [19] as an example. Our graph data structure includes 4 arrays:

- *OneHopAdj* (resp. *TwoHopAdj*): the concatenation of the one-hop (resp. two-hop) adjacency lists;
- *OneHopOffset* (resp. *TwoHopOffset*): the start location of the neighbor list of vertex i in *OneHopAdj* (resp. *TwoHopAdj*);

Task Data Structure. The real information of each task is prepared in CSR format (as shown in Fig. 4). We keep these tasks in the global memory compactly as 6 arrays:

- *taskOffset*: *taskOffset[i]* represent the start location of task i in the *vertices* array;
- *vertices*: all vertices in $\langle S, ext(S)\rangle$ of each task;
- *label*: for vertex *vertices[i]*, *label[i]* identifies its corresponding status. 0 means *vertices[i]* is in S, 1 means it is in *ext(S)*, 2 means being covered (see IV-G), and -1 means being pruned;
- *indeg*: the number of neighbors that *vertices[i]* has in S, i.e. *indeg[i]* = $|N_S(vertices[i])|$;
- *exdeg*: the number of neighbors that *vertices[i]* has in $ext(S)$, i.e. *exdeg[i]* = $|N_{ext(S)}(vertices[i])|$;
- *lvl2adj*: *lvl2adj[i]* = number vertices in *ext(S)* which are within 2-hops from *vertices[i]*;

As shown in Fig. 4, suppose we have 3 tasks from a fully connected graph {a,b,c,d}. T_1 has $\langle S_1, ext(S_1)\rangle = \{(a, b), (c, d)\}$; T_2 has $\langle S_2, ext(S_2)\rangle = \{(b, c), (d)\}$; T_3 has $\langle S_3, ext(S_3)\rangle = \{(c, d), \emptyset\}$. Then, these 3 tasks' required information will be represented as the 6 arrays in Fig. 4. This array-based data structure is ideal as direct index access allows each warp to read its tasks in parallel.

C. GPU Algorithm

Host Program. The host program is presented in Algorithm 1. Line 1 loads the input graph into the main memory utilizing the CSR structure. Line 2 then executes an initial round of cover pruning (see Section IV-G). The covered vertices are exempt from spawning in Line 3 (status being marked in the label array). This ensures each vertex initiates a task following the specific data structure in Fig. 4.

Given that tasks correspond to subtrees spawned from the nodes in the set-enumeration tree as Fig. 2, it implied that the number of tasks is quite small initially. Consequently, it is not optimal to offload these preliminary tasks to the GPU, as their limited quantity would not fully utilize the GPU's computing power, leaving most of the GPU cores idle. Therefore we have variable *cpu_round* initialized to 0 in Line 4 to track the number of rounds executed on the CPU. With a hybrid CPU-GPU implementation strategy, the tasks will be transitioned to GPU processing only after exceeding the number of rounds specified by hyperparameter τ_{cpu}. Lines 5–8 execute the CPU *expand(t)* function as Quick+ introduced in Section III-A.

From Line 9 onwards, the program shifts to GPU execution. Line 9–10 prepare the required task data structures as Fig. 4 on GPU and copy the tasks from the CPU main memory to the GPU memory. As shown in Fig. 5 the task are stored in *TaskList* and *TaskBuffer* (an auxiliary container to control task spawning speed, cf. Section IV-D). In Lines 11–16, each task is processed by one warp until both the *TaskList* and *TaskBuffer* on GPU are empty. Notably, Line 12 invokes the *gpu_expand(t)* kernel function, where each warp expands an individual task into new tasks, applying pruning techniques to narrow down the search space for each new task. These new tasks are temporarily stored within the *WarpTaskBuffer* (see Fig. 5), to avoid the inefficiency of duplicated two-pass computations (detailed in Section IV-D). Upon completion of a round, once all tasks have been written to the *WarpTaskBuffer* by warps, Line 13 triggers the *transfer()*. This function transfers the newly expanded tasks back to the *TaskList*, or to the *TaskBuffer* if capacity constraints necessitate. Line 14 ensures the replenishment of tasks from *TaskBuffer* to *TaskList* to maintain warp occupancy. Throughout the execution of *gpu_expand(t)*, all valid quasi-cliques are stored in the GPU global memory in *ResultList*. Line 16 transfers results from *ResultList* to the host.

Kernel Program. Algorithm 2 shows an *expand* kernel, where the tasks are distributed to warps for further expansion and pruning. Initially, to read tasks from *TaskList* in parallel by warps on GPU, we maintain a cumulative counter, *global_count*, to track the next task to be processed. In

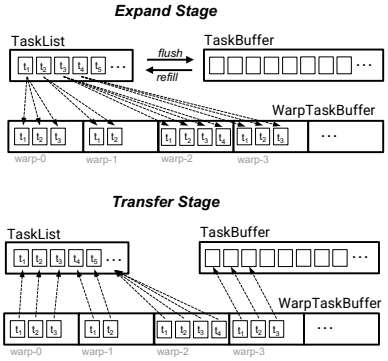


Fig. 5. Expand and Transfer

Lines 1–2, each warp localizes the next task in *TaskList* and increments the *global_count* by one. Lines 3–23 repeatedly expand current task, employ pruning strategies to reduce search space, verify the validity, and retrieve the next tasks.

Specifically, Lines 5–8 perform the lookahead pruning to verify if the entire $S \cup ext(S)$ collectively satisfies the criteria for a γ-quasi-clique. Subsequently, Line 11 move one vertex v from the candidate set $ext(S)$ to S. Given the constraint $\gamma \geq 0.5$, all the vertices must be within 2-hops to form a γ-quasi-clique (see Section II). Line 12 removes vertices from the candidate set that are not within 2-hops of v. To fully leverage the high-speed shared memory, the new task with $\langle S', ext(S')\rangle$ will be stored in shared memory if $|S'| < \tau_{shared}$, or stored in global memory. τ_{shared} is a hyperparameter defined by users according to their GPU hardware. Lines 13–15 engage in pruning the new tasks, employing CUDA warp-level primitives to implement the specified pruning criteria. The pruning rules are classified into two categories: Type I aims to eliminate unpromising vertices from the candidate set, whereas Type II can terminate exploration on the current task as demonstrated in Lines 16–17 and detailed within Section IV-G. Lines 18–20 store new tasks in *WarpTaskBuffer* if they can be expanded further and check if the current S' is a γ-quasi-clique. Once all the candidates in this task have been explored, Lines 21–22 will localize the next task.

Another kernel function *transfer()* only transfers tasks and will be introduced in the Single-Pass Expansion Approach.

D. Approach

Task Flush and Refill. To manage memory usage, the system control the number of tasks to be expanded concurrently. If all available tasks are expanded at every level in the set-enumeration tree, memory usage may explode on big graphs. But, in terms of efficiency, maintaining a sufficient number of tasks in *TaskList* is also crucial to keep all warps busy. To control the task expansion rate, we introduce an expansion threshold τ_{expand}, which specifies the number of tasks planned for each level. During expansion, if the number of generated tasks exceeds the τ_{expand}, we flush the excess tasks from the *WarpTaskBuffer* to the *TaskBuffer* (see Fig. 5). Before expanding, if the number of tasks in *TaskList* is less than the τ_{expand}, we replenish it by transferring tasks from the

Algorithm 1 GPU-Based MQC Algorithm: Host Program

1: Load Graph G into the host memory
2: $u \leftarrow \arg\max_{v \in V} \deg(v)$ ▷ Select highest-degree vertex
3: Spawn tasks from $V - N(u)$ and save in task list T
4: $cpu\_round \leftarrow 0$
5: **while** $cpu\_round > \tau_{cpu}$ **do**
6: **for each** task $t \in T$ **do**
7: $expand(t)$
8: $cpu\_round \leftarrow cpu\_round + 1$
9: Allocate $TaskList$ in device memory
10: Copy task list T from host to the device $TaskList$
11: **while** $TaskList \neq \emptyset$ **and** $TaskBuffer \neq \emptyset$ **do**
12: Launch kernel $gpu\_expand(t)$
13: Launch kernel *transfer()*
14: **if** $|TaskList| < \tau_{expand}$ **and** $TaskBuffer \neq \emptyset$ **then**
15: Refill from $TaskBuffer$ to $TaskList$
16: Save $ResultList$ in file

Algorithm 2 Kernel Function `gpu_expand()`

1: **if** $LANE\_ID = 0$ **then**
2: $loc \leftarrow$ atomicAdd($global\_count$, 1)
3: **repeat**
4: $WarpTask \leftarrow TaskList[loc]$
5: **if** $G(S \cup ext(S))$ is a γ-quasi-clique **then**
6: Append $G(S \cup ext(S))$ to $ResultList$
7: **if** $LANE\_ID = 0$ **then**
8: $loc \leftarrow$ atomicAdd($global\_count$, 1)
9: **continue**
10: **for each** v **in** $ext(S)$ **do**
11: $S' \leftarrow S \cup v, \quad ext(S) \leftarrow ext(S) - v$
12: $ext(S') \leftarrow ext(S) \cap \{v$'s 2-hop-neighbors$\}$
13: Warp-Level Degree-Based Pruning
14: Warp-Level Lower/Upper-Bound Based Pruning
15: Warp-Level Critical Vertex Pruning
16: **if** any Type II pruning on S' is triggered **then**
17: **continue**
18: Append new task $T_{\langle S', ext(S')\rangle}$ to *WarpTaskBuffer*
19: **if** S' is a γ-quasi-clique **then**
20: Append S' to $ResultList$
21: **if** $LANE\_ID = 0$ **then**
22: $loc \leftarrow$ atomicAdd($global\_count$, 1)
23: **until** $loc \geq |TaskList|$

TaskBuffer. Table IV in Section VI illustrates that τ_{expand} significantly impacts both the speed and memory consumption.

Hybrid CPU-GPU Approach. The size of tasks generated at lower levels of expansion is generally large, as less pruning has been done to each task. There will be a small amount of tasks, but each one will be exceptionally intensive. Directly running this small amount of heavy tasks on the GPU will introduce two issues: (1) workload imbalance, where most warps remain idle while a few are busy mining those heavy tasks (as discussed in Section IV-C), and (2) a significant increase in space demands due to these large early-stage tasks

generating a huge amount of sub-tasks.

To address this problem, we consider running the first few levels on the CPU before moving to the GPU to minimize the *WarpTaskBuffer* size required by each warp. As mentioned in Section IV-C, users can control the transition point from CPU to GPU processing by adjusting the τ_{cpu} hyperparameter. We found switching after the second level gives the best performance for most cases. The hybrid CPU-GPU approach significantly decreases the memory required for expansion on the GPU allowing for the program to run on larger graphs.

Single-Pass Expansion Approach. Numerous GPU-based algorithms [21] take a two-pass approach when multiple threads need to write an unknown amount of heterogeneous output in parallel. The underlying logic of this two-pass method involves determining the size of each element during the first pass, followed by parallel output writing in the second pass. This principle is particularly applicable to parallel MQC mining. To get the writing location, it is essential to obtain the number and size of new tasks by executing a preliminary *expand* pass. Subsequently, a second *expand* pass is necessary to generate and store these new tasks in parallel.

As shown in Fig. 5, we developed a novel one-pass strategy utilizing an exclusive scan operation facilitated by the additional data structure *WarpTaskBuffer*. This structure preallocates buffers for each warp to write their partial results to. When the partial results generated by all warps are ready, the second kernel *transfer()* is launched to transfer the data from *WarpTaskBuffer* to *TaskList* or *TaskBuffer* (depending on the capacity of *TaskList*). Then we run an exclusive scan to assess the size of tasks to be recorded in the *TaskList* and accurately determine the offsets. With the offsets, each warp will use its threads to transfer the data it generated to the *TaskList* or *TaskBuffer* in parallel as shown in Fig. 5.

The one-pass approach requires the additional calculation and synchronization of the *transfer()* kernel, but we have observed that this takes minimal time compared to the general *gpu_expand()* kernel call. As shown in our experiment, avoiding duplicate calculations leads to significant time savings.

E. Task Scheduling Approach

In many GPU-based graph mining algorithms [21], [22], an equal distribution of jobs across each computing unit is implemented. However, this approach of evenly allocating tasks to each warp leads to workload balancing issues when mining MQC. As demonstrated in [2], the runtime of MQC tasks varies significantly, and it is difficult to predict each task's runtime because of the complex pruning rules involved.

In this study, we implement a dynamic task scheduling approach, in contrast to the traditional method of statically allocating an equal number of tasks to each warp in advance. Instead, warps proactively fetch new tasks from *TaskList* upon completing their current ones. Our experiments indicate that this dynamic method significantly enhances performance.

F. Vertex Set in Shared Memory

Accessing shared memory, though size-limited, can be approximately 10 to 100 times faster than accessing global memory. To fully utilize the shared memory, we allocate a buffer of size τ_{shared} called *SharedVertices* for each warp. During task expansion, newly generated tasks smaller than τ_{shared} will be stored in *SharedVertices*.

G. Pruning Rules

Our program utilizes the 6 pruning rules from Quick [3] to greatly reduce the search space of a graph.

- Diameter Pruning: given $\gamma \geq 0.5$, each time a vertex is added to the S from $ext(S)$, all vertices not within 2 hops of the new vertex can be eliminated.
- Degree-Based Pruning: it considers the minimum number of neighbors each vertex must have in the clique and whether that requirement is still achievable. This pruning rule applies whenever the degrees of a vertex change.
- Cover Pruning: one vertex may have a strong connection with a group of other vertices, and they cannot form an MQC without including this key vertex. This vertex is referred to as a cover vertex in Quick [3], and the group of vertices it covers can be excluded from expansion.
- Lookahead Pruning: at the beginning of the expansion, we check whether the entire set forms a quasi-clique.
- Upper-lower Bound Pruning: it calculates the upper and lower bound on the vertices that can be added to S while maintaining the potential to form an MQC. These refined bounds enable further degree-based pruning.
- Critical Vertex Pruning: A critical vertex in S requires all neighbors in the candidate set to be included to form an MQC. We identify these vertices and add their neighbors.

More details of pruning rules can be found in Quick [3]. These pruning rules all complement each other and are performed repeatedly. Without all these rules, analyzing larger graphs is impossible.

Example: Degree-Based Pruning. Adapting these pruning techniques on GPU necessitated innovative memory-management strategies. Due to space limitations, we cannot provide the GPU implementation details for each pruning technique. Instead, we use degree-based pruning as an example.

Degree-based pruning occurs during expansion after vertex v is added to S, removing all vertices in $ext(S)$ not within 2-hops of v. The *indeg* and *exdeg* (as in Fig. 4) are updated after removal. Vertices that no longer meet the degree criteria are removed, triggering further degree updates and pruning until no vertices can be removed. To perform this efficiently, we initialize auxiliary structures as shown in Fig. 6: *RemainingVertices* (stores unpruned vertices) and *RemovedVertices* (stores pruned vertices). Degree updates are performed via an intersection after vertex removal by either:

$$\forall v \in RemainingVertices,$$
$$exdeg(v) \leftarrow exdeg(v) - |N_G(v) \cap RemovedVertices|$$

or

$$exdeg(v) \leftarrow |N_G(v) \cap RemainingVertices|$$

Kernel profiling shows the intersection is the most computationally intensive part of the program due to the large

component sizes. Each thread handles the degree calculation above for one vertex in the task. We optimize the intersection by dynamically selecting the most efficient method above based on the sizes of *RemainingVertices* and *RemovedVertices*, favoring the smaller set for faster intersection. Additionally, we preprocess and sort the adjacency list, enabling binary search for neighbors in the remaining or removed sets.

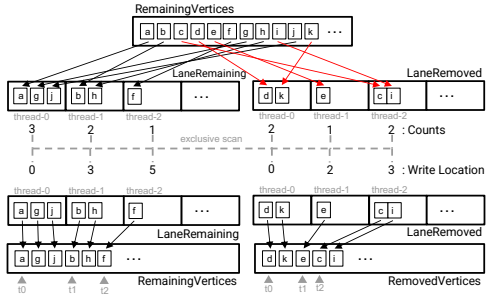


Fig. 6. Parallel Degree-Pruning on GPU

After the intersection, we obtain an updated *RemainingVertices* (top of Fig. 6) with revised vertex degrees. Using these, each thread applies degree-based pruning to one vertex. Directly writing results to *RemainingVertices* and *RemovedVertices* risks race conditions. To avoid this, we use per-thread buffers: *ThreadRemaining* and *ThreadRemoved*. As indicated by the top arrows in Fig. 6, each thread writes to its own buffers and tracks counts of remaining and removed vertices in registers. Then an exclusive scan across the warp—using CUDA warp-level primitives—determines each thread's write offset (e.g., t0, t1, t2 in Fig. 6) into *RemainingVertices* and *RemovedVertices*. Threads then merge results from local buffers into the global structures. This pruning loop repeats until no more vertices can be removed.

V. Distributed Memory

Real-life graphs are typically large and complex, making a single GPU insufficient for processing. To address this, we developed a multi-GPU version of cuQC to boost the performance. The core concept involves redistributing tasks: once a GPU completes its assigned tasks, it receives additional tasks from others still in progress.

Distributed Program. The distributed program is presented in Algorithm 3. Line 1 represents the first 10 lines of Algorithm 1, which involve loading the data, performing CPU expansion, and transferring the data to the GPU. Notably, the distributed version still follows the hybrid CPU-GPU design. Enough tasks will be spawned after the CPU stage to feed the GPUs. To balance the workload between multiple GPUs, these tasks are distributed among different GPU nodes in a strided manner, as the early spawned tasks are likely to be larger due to having more candidates. For example, if there are 16 tasks with IDs ranging from 0 to 15 to be assigned among 4 GPUs, GPU-0 receives tasks t_0, t_4, t_8, t_{12}, GPU-1 receives tasks t_1, t_5, t_9, t_{13}, and so on.

Lines 3–8 perform the local GPU expansion of tasks until completion. Specifically, Line 4 corresponds to lines 12–16 of Algorithm 1, which generate the next level of tasks from the current level. Unlike the previous algorithm, at the end of each level, Line 5 checks if *TaskBuffer* exceeds τ_{help}. If so, Line 6 invokes the *send_work()* method, which attempts to send idle tasks to another process and returns whether it is successful. If it is successful, τ_{send} of tasks from *TaskBuffer* will be sent to the receiving process's *TaskBuffer*. The hyperparameter τ_{help} and τ_{send} ensure tasks are sent only when their volume justifies the network communication overhead, which is set by the user based on network speed. Line 7 verifies sending success, and Line 8 updates *TaskBuffer* size accordingly.

Algorithm 3 Distributed Memory Algorithm

```
1:  Execute lines 1-10 from Algorithm 1
2:  while !all_GPUs_free() do
3:      while TaskList ≠ ∅ and TaskBuffer ≠ ∅ do
4:          Execute lines 12-16 from Algorithm 1
5:          if TaskBuffer > τ_help then
6:              send ← send_work()
7:              if send then
8:                  Update TaskBuffer
9:      broadcast_free(rank)
10:     receive ← receive_work()
11:     if receive then
12:         Update TaskBuffer
13:         Refill TaskBuffer to TaskList
14:         broadcast_received(rank)
15: Save ResultList to file
```

Continuing from Line 9, all local tasks have been completed, and the focus shifts to obtaining tasks from a process that still has tasks. Initially, a broadcast message is sent to all processes, indicating that this process is now free. Line 10 invokes the *receive_work()* method, which attempts to receive work from another process and returns a success status. If tasks are received successfully, Lines 12–14 update the *TaskBuffer*, fill *TaskList* from *TaskBuffer*, and broadcast to all other processes that this process is no longer free. The communication between multi-GPUs is discussed as follows.

Message Passing. To ensure that one node will send its idle tasks to only one other node, we have designed the communication to follow a "Three-way Handshake" approach. As shown in Fig. 7, when a process becomes free, it broadcasts its status as f to all other processes. The free process will then continuously check for messages from other processes. It will wait until either all processes' statuses become f (indicating the entire program is finished) or one process sends an r (indicating a busy node is requesting help). The free process will respond to the busy node with a c message, asking for confirmation. This confirmation is necessary because, due to the asynchronous design, the requesting process might have completed all its tasks by the time the confirmation message arrives. If the busy process still has tasks, it will reply with C to one idle process, initiating the transfer. When the idle process has received the work, it will broadcast t (taken work), declaring that it is no longer free. If multiple free processes have sent a c, the busy process will send a D (decline) to

the others, indicating that it is declining their help, prompting them to continue looking for other requesting processes.

This system is designed to be asynchronous, with all processes being non-blocking except those involved in the actual sending and receiving of tasks. This blocking communication is highlighted in red in Fig. 7. Messages containing vertex data are efficiently transmitted using *MPI_Datatype* in a single transaction, avoiding the need to send different variables in multiple messages. These designs ensure maximal GPU utilization and guarantee excellent scalability as the number of GPUs increases, as demonstrated in the following experiments.

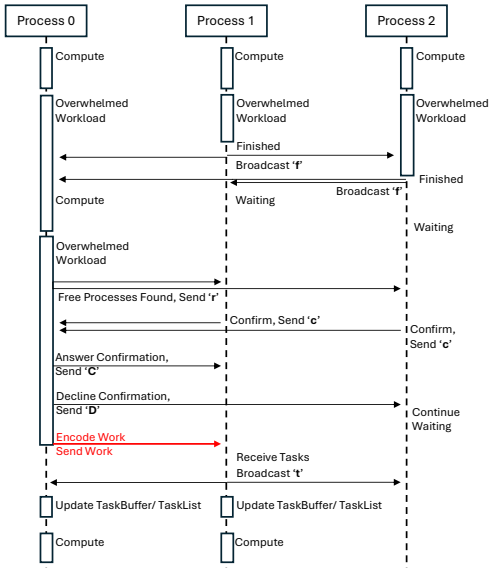


Fig. 7. Distributed GPUs Communication

VI. EXPERIMENTS

In this section, we conduct comprehensive experiments to evaluate the performance of cuQC.

A. Experimental Setup

Platform. We evaluate our GPU implementations on an NVIDIA A100 GPU [23] with 108 streaming multiprocessors (SMs) and 80 GB of global memory. For the comparison, we run the other CPU-based MQC algorithm T-thinker on a Linux server with AMD EPYC 7763 64-Core Processor @ 2.45GHz CPU cores.

Datasets. We comprehensively evaluate our cuQC algorithm and baselines on 21 public graph datasets with varying sizes and densities, from SNAP [24], Network Repository [25], the GraphChallenge [26], and NCBI [27]. As shown in Table I, the datasets are listed in ascending order of the number of vertices. These datasets span numerous categories, including (1) social networks such as Ego-Facebook, LastFM, FB-Pages, Brightkite, Gowalla, YouTube, Hyves, Flixster, Livejournal, and Konect; (2) collaboration networks such as Ca-GrQc, HepPh, AstroPh, CondMat, Citeseer, DBLP; (3) biological network CX_GSE1730; (4) email communication network Enron; (5) a co-purchasing network Amazon; (6) an internet topology Skitter; and (7) a protein Kmer.

TABLE I
GRAPH DATASETS

Dataset	|V|	|E|	|E| / |V|	Max Degree
CX_GSE1730	998	5,096	5.11	197
Ego-Facebook	4,039	88,234	21.85	1,045
Ca-GrQc	5,242	14,496	2.77	81
LastFM	7,624	27,806	3.65	216
HepPh	12,008	118,521	9.87	491
AstroPh	18,772	198,110	10.55	504
CondMat	23,133	93,497	4.04	280
Enron	36,692	183,831	5.01	1,383
FB-Pages	50,516	819,306	16.22	1,469
Brightkite	58,228	214,078	3.68	1,134
Gowalla	196,591	950,327	4.83	14,730
Citeseer	227,320	814,134	3.58	1,372
DBLP	317,080	1,049,866	3.31	343
Amazon	334,863	925,872	2.76	549
YouTube	1,134,890	2,987,624	2.63	28,754
Hyves	1,402,673	2,777,419	1.98	31,883
Skitter	1,696,415	11,095,298	6.54	35,455
Flixster	2,523,387	7,918,801	3.14	1,474
Livejournal	4,033,138	27,933,062	6.93	2,651
Konect	59,216,212	92,522,014	1.56	4,960
Kmer	67,716,231	69,389,281	1.02	35

Compared Algorithms. Since there is no existing GPU-accelerated algorithm for MQC, we compare cuQC to CPU-oriented algorithms, including the recent serial version Quick [3], and the cutting-edge parallel MQC algorithm, i.e., parallel Quick+ on T-thinker [18]. By default, cuQC sets the values for τ_{cpu} (rounds ran on CPU) to 2 and uses dynamic scheduling. We also implement other variants to evaluate the techniques proposed in this paper, which will be detailed in the corresponding experiments.

B. Overall Evaluation

We configure a kernel grid to have 216 thread blocks, with each block comprising 1,024 threads. Table II compares the running times of the serial Quick, parallel CPU-based T-thinker, and our GPU-based program cuQC on real-world datasets. The symbol "-" indicates that the program could not complete execution within 24 hours. The experimental results demonstrate that our cuQC can achieve a remarkable 3,992x speedup compared to the serial Quick algorithm when evaluated on the AstroPh dataset. This exceptional acceleration shows the significant performance gains attainable by cuQC through its effective utilization of the massively parallel computing capabilities of the GPU.

Our cuQC achieve an impressive 179x speedup over the parallel CPU-based T-thinker system (see Table II). This acceleration is significantly observed in dense graphs, exemplified by dataset Ego-Facebook with an average degree of 21.85 and AstroPh with an average degree of 10.55. During the exploration of the set-enumeration tree, these dense graphs spawn a multitude of computationally intensive tasks that cannot be easily pruned. Our GPU system is well-designed for efficiently performing the pruning calculations associated with these demanding tasks, enabling cuQC to outperform both CPU-based competitors by orders of magnitude on dense graph datasets. Additionally, T-thinker flushes intermediate tasks to disk to maintain memory bounded. After two hours run on the AstroPh and Ego-Facebook datasets, we had to terminate the T-thinker system, as generated files were about to fill our 2 TB disk.

Due to space limitations, we are unable to present experiments for all datasets. Instead, we represent the datasets by

TABLE II
OVERALL EVALUATION

Dataset	γ	τ_{size}	#{MQC}	Quick (ms)	T-thinker (ms)	cuQC (ms)
CX_GSE1730	0.9	30	1,602	191	235	120
Ego-Facebook	0.95	103	2	-	-	230,940
Ca-GrQc	0.8	10	43,399	366	242	199
LastFM	0.75	20	23,319	7,405	5,465	199
HepPh	0.95	71	5	-	-	228
AstroPh	0.8	54	54,772	906,245	40,754	227
CondMat	0.8	15	4,396	835	625	199
Enron	0.9	23	200	127,926	12,813	3,284
FB-Pages	.9	24	17	1,894,457	2,174,848	440,164
Brightkite	0.9	50	1,361	500,637	25,319	374
Gowalla	0.9	30	11	1,073,253	65,037	14,108
Citeseer	0.9	60	26,312	-	1,494	595
DBLP	0.9	73	2	1,540	1660	158
Amazon	0.5	12	13	1,028	1,230	377
YouTube	0.9	18	274	-	898,070	716,804
Hyves	0.9	22	1,480	190,655	109,564	13,846
Skitter	0.95	53	96	-	-	73,875
Flixster	.9	50	49	-	-	53,234
Livejournal	.95	196	56,057	-	-	63,213
Konect	.5	15	25,392	-	-	13,568,398
Kmer	0.5	10	63	49,752	48,215	28,311

size. For small graphs ($|V| < 100,000$), we chose Ego-Facebook. For medium graphs ($100,000 \leq |V| \leq 1,000,000$), we selected Gowalla. For large graphs ($|V| > 1,000,000$), we chose Skitter. These graphs were chosen for their high $|V|/|E|$ ratios, ensuring they would present non-trivial challenges. *Some experiments cannot use these representative data, as explained in the corresponding sections.*

C. Effect of Optimizations

Static Scheduling v.s. Dynamic Scheduling. To analyze the impact of task scheduling strategies from Section IV, we designed two variants of cuQC: a static scheduling version and a dynamic scheduling version. Table III illustrates that the dynamic scheduling strategy significantly improve the performance, as it promotes workload balancing by assigning the next available task to an idle warp on the GPU immediately, taking full advantage of the GPU's computational resources.

TABLE III
TASK SCHEDULING

Dataset	Static (ms)	Dynamic (ms)	Speedup
Ego-Facebook	254,173	243,744	1.04
Gowalla	17,770	14,293	1.24
Skitter	79.897	74,517	1.07

D. Hyperparameter Analysis

Our system has a configurable setting τ_{expand}, which approximately determines the number of tasks handled by each warp per round. This setting indirectly determines the size of the *WarpTaskBuffer* and *TaskBuffer* containers. As shown in Table IV, this hyperparameter impacts both runtime and memory usage. For instance, by increasing τ_{expand} from $6,912$ ($Warp\#/Block \times BlockNum$) to $691,200$ on Ego-Facebook, the runtime decreases from $582{,}672$ ms to $243{,}529$ ms, while the memory usage increases from $2{,}003$ MB to $18{,}261$ MB.

Increasing τ_{expand} reduces time by activating more tasks in each level kernel call. It decreases the data transfers frequency between the *WarpTaskBuffer* and *TaskBuffer*, and more active tasks enhance the ability to balance workloads using the dynamic task scheduling strategy. However, it will also increase memory usage since more tasks will generate sub-tasks in the same level kernel call.

We aim to run all graphs with a τ_{expand} of 100, but for the Youtube graph, we use a τ_{expand} of 10 to bound memory. This hyperparameter enhances the adaptability across different GPUs. For GPUs with limited memory capacity, users can still scale to big graphs by tuning the τ_{expand} setting accordingly. This flexibility enables our system to accommodate a wide range of GPU configurations, facilitating efficient quasi-clique mining on diverse hardware platforms while balancing runtime and memory requirements.

TABLE IV
EFFECT OF EXPAND THRESHOLD

Dataset	τ_{expand}	Time (ms)	Memory (MB)
Ego-Facebook	6,912	582,672	2,003
	13,824	416,515	2,313
	69,120	277,471	4,627
	345,600	247,587	11,327
	691,200	243,529	18,261
Gowalla	6,912	17,173	29,975
	13,824	15,125	29,975
	69,120	14,222	29,975
	345,600	14,492	29,983
	691,200	14,471	29,987
Skitter	6,912	241,715	34,849
	13,824	183,490	35,367
	69,120	106,353	36,509
	345,600	75,355	38,523
	691,200	74,636	42,031

E. Ablation Study

In this experiment, we evaluate the effectiveness of the pruning techniques on the GPU. We selected smaller graphs for these tests because, without pruning, the search space becomes too large to handle on larger graphs. Moreover, diameter and degree-based pruning are fundamental to the program and are already included in all algorithms. We implemented a baseline mining algorithm that does not use most pruning techniques described in Section IV. We then add one of the pruning techniques to the baseline GPU algorithm. Table V presents the running time of the baseline algorithm and pruning techniques on LastFM and Condmat. These results highlight the necessity of assembling all proposed techniques on the GPU, as no single pruning technique exhibits consistent effectiveness across all datasets. For instance, the lookahead technique does not provide significant benefits on the LastFM dataset. However, it achieves a speedup ratio of 24.67 on the Condmat dataset. This implies that the effectiveness of the pruning techniques is dependent on the characteristics of the underlying datasets.

TABLE V
EFFECTIVENESS OF PRUNING TECHNIQUES ON GPU

Dataset	Algorithm	Time (ms)	Speedup
LastFM	Baseline	4,113	-
	Baseline + Lookahead	3,616	1.14
	Baseline + UpperLower	1,851	2.22
	Baseline + CriticalVertex	2,002	2.05
	cuQC	779	5.28
Condmat	Baseline	27,429	-
	Baseline + Lookahead	1,112	24.67
	Baseline + UpperLower	27,543	1.00
	Baseline + CriticalVertex	26,802	1.02
	cuQC	1,062	25.83

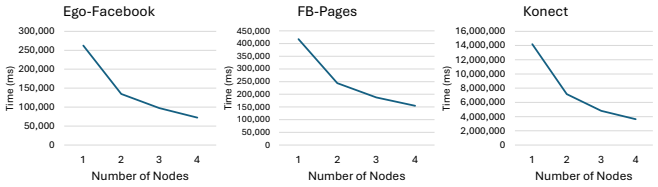
F. Distributed Memory

We evaluate our distributed GPU implementation with 4 NVIDIA A100 GPUs [23] (the maximum in our cluster).

Datasets. Compared to the single-node version, the distributed version is designed for large graphs which have sufficient work to utilize all GPUs. We run tests on three large, high-density datasets: Ego-Facebook, FB-Pages, and Konect [24], [25].

Evaluation. Table VI reports the scalability as we vary the number of GPUs as 1, 2, 3, and 4. We can see that additional machines generally improves the performance. Perfect scalability is unattainable due to some parts of the program running on the CPU and the communication overhead between GPUs. Our results show that near-ideal scalability can be achieved on graphs with substantial workloads. For the 4 GPU version on the Konect graph, we achieved a speedup of 3.9 times compared with a single node.

TABLE VI
SCALABILITY



Effect of Help Threshold. In our program, we have a configurable setting, τ_{help}, which specifies the number of remaining tasks at which processes should begin distributing work to other GPUs. A lower τ_{help} means more tasks can be balanced to other idle GPUs from a busy one, but it also induces more communication overhead. Note that our experiment starts with τ_{help} set at $6,912$ to ensure at least one task is left for each warp. As shown in Table VII, a τ_{help} of the minimum $6,912$ is ideal. This is due to the extremely fast intra-node communication on our server, making messaging times for these larger graphs negligible compared to computation time.

TABLE VII
EFFECT OF HELP THRESHOLD

Ego-Facebook		*FB-Pages*		*Konect*	
τ_{help}	Time (ms)	τ_{help}	Time (ms)	τ_{help}	Time (ms)
6,912	148,201	6,912	155,779	6,912	3,640,231
13,824	150,674	13,824	156,903	13,824	3,657,240
69,120	205,089	69,120	156,329	69,120	3,656,189
345,600	205,016	345,600	160,851	345,600	3,642,673
691,200	205,100	691,200	160,143	691,200	3,639,677

VII. CONCLUSION

We have designed the first highly efficient MQC system on the GPU and a corresponding distributed version. Mining MQC on big graphs using the GPU faces serious challenges, including large memory requirements, thread divergence, and severe load imbalance. To address these problems, novel GPU-aware data structures and optimization techniques are designed on our cuQC. Our experiments show that cuQC significantly outperforms traditional serial CPU solutions and is also faster than parallel CPU solutions.

ACKNOWLEDGMENTS

This work was supported by NSF CRII-2245792.

REFERENCES

[1] Y. Fang, K. Wang, X. Lin, and W. Zhang, "Cohesive subgraph search over big heterogeneous information networks: Applications, challenges, and solutions," in *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*.

[2] G. Guo, D. Yan, M. T. Özsu, Z. Jiang, and J. Khalil, "Scalable mining of maximal quasi-cliques: An algorithm-system codesign approach," *Proc. VLDB Endow.*, vol. 14, no. 4, pp. 573–585, 2020.

[3] G. Liu and L. Wong, "Effective pruning techniques for mining quasi-cliques," in *ECML/PKDD*, ser. Lecture Notes in Computer Science, W. Daelemans, B. Goethals, and K. Morik, Eds. Springer, 2008.

[4] M. Bhattacharyya and S. Bandyopadhyay, "Mining the largest quasi-clique in human protein interactome," in *2009 International Conference on Adaptive and Intelligent Systems*. IEEE, 2009, pp. 194–199.

[5] H. Matsuda, T. Ishihara, and A. Hashimoto, "Classifying molecular sequences using a linkage graph with their pairwise similarities," *Theor. Comput. Sci.*, vol. 210, no. 2, pp. 305–325, 1999.

[6] J. Li, X. Wang, and Y. Cui, "Uncovering the overlapping community structure of complex networks by maximal cliques," *Physica A: Statistical Mechanics and its Applications*, vol. 415, pp. 398–406, 2014.

[7] J. Hopcroft, O. Khan, B. Kulis, and B. Selman, "Tracking evolving communities in large linked networks," *Proceedings of the National Academy of Sciences*, vol. 101, no. suppl 1, pp. 5249–5253, 2004.

[8] C. Wei, A. Sprague, G. Warner, and A. Skjellum, "Mining spam email to identify common origins for forensic application," in *ACM Symposium on Applied Computing*, R. L. Wainwright and H. Haddad, Eds. ACM, 2008, pp. 1433–1437.

[9] S. Sheng, B. Wardman, G. Warner, L. Cranor, J. Hong, and C. Zhang, "An empirical analysis of phishing blacklists," in *6th Conference on Email and Anti-Spam (CEAS)*. Carnegie Mellon University, 2009.

[10] S. Sanei-Mehri, A. Das, and S. Tirthapura, "Enumerating top-k quasi-cliques," in *IEEE BigData*. IEEE, 2018, pp. 1107–1112.

[11] G. Pastukhov, A. Veremyev, V. Boginski, and O. A. Prokopyev, "On maximum degree-based γ-quasi-clique problem: Complexity and exact approaches," *Networks*, vol. 71, no. 2, pp. 136–152, 2018.

[12] J. Pei, D. Jiang, and A. Zhang, "On mining cross-graph quasi-cliques," in *SIGKDD*. ACM, 2005, pp. 228–238.

[13] D. Jiang and J. Pei, "Mining frequent cross-graph quasi-cliques," *ACM Trans. Knowl. Discov. Data*, vol. 2, no. 4, pp. 16:1–16:42, 2009.

[14] Z. Zeng, J. Wang, L. Zhou, and G. Karypis, "Coherent closed quasi-clique discovery from large dense graph databases," in *SIGKDD*. ACM, 2006, pp. 797–802.

[15] A. Quamar, A. Deshpande, and J. Lin, "Nscale: neighborhood-centric large-scale graph analytics in the cloud," *The VLDB Journal*, 2014.

[16] C. H. C. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga, "Arabesque: a system for distributed graph mining," in *SOSP*, 2015, pp. 425–440.

[17] H. Chen, M. Liu, Y. Zhao, X. Yan, D. Yan, and J. Cheng, "G-miner: an efficient task-oriented graph mining system," in *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*, R. Oliveira, P. Felber, and Y. C. Hu, Eds.

[18] J. Khalil, D. Yan, G. Guo, and L. Yuan, "Parallel mining of large maximal quasi-cliques," *VLDB J.*, vol. 31, no. 4, pp. 649–674, 2022. [Online]. Available: https://doi.org/10.1007/s00778-021-00712-2

[19] M. Valiyev, "Graph storage: How good is csr really?" *dated Dec*, vol. 10, p. 8, 2017.

[20] J. L. Greathouse and M. Daga, "Efficient sparse matrix-vector multiplication on gpus using the CSR storage format," in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2014, New Orleans, LA, USA, November 16-21, 2014*, T. Damkroger and J. J. Dongarra, Eds.

[21] Y. Wei, W. Chen, and H. Tsai, "Accelerating the bron-kerbosch algorithm for maximal clique enumeration using gpus," *IEEE Trans. Parallel Distributed Syst.*, vol. 32, no. 9, pp. 2352–2366, 2021.

[22] L. Xiang, A. Khan, E. Serra, M. Halappanavar, and A. Sukumaran-Rajam, "cuts: scaling subgraph isomorphism on distributed multi-gpu systems using trie based data structure," in *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2021, St. Louis, Missouri, USA, November 14-19, 2021*, B. R. de Supinski, M. W. Hall, and T. Gamblin, Eds. ACM, 2021, p. 69.

[23] "NVIDIA A100 Tensor Core GPU," https://www.nvidia.com/en-gb/data-center/a100/.

[24] J. Leskovec and A. Krevl, "Snap datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, 2014.

[25] R. Rossi and N. Ahmed, "The network data repository with interactive graph analytics and visualization," http://networkrepository.com, 2015, accessed: 2025-08-28.

[26] "Kmer," https://graphchallenge.s3.amazonaws.com/synthetic/gc6/U1a.tsv.

[27] "CX_GSE1730," https://www.ncbi.nlm.nih.gov/geo/query/acc.cgi?acc=GSE1730.